# EXHIBIT C

**Exhibit A-15**

**Invalidity of U.S. Patent No. 6,851,115 ("'115 Patent")**
**by *Kiss***

As shown in the claim chart below, the asserted claims of the '115 patent are invalid under at least 35 U.S.C. §§ 102(e)[1] as anticipated by U.S. Patent No. 6,484,155 ("*Kiss*") filed on July 21, 1999, issued on November 19, 2002, with an earliest potential priority date of July 21, 1998, and/or are invalid under 35 U.S.C. § 103 as obvious in view of *Kiss* or in combination with the reference(s) specifically identified in the following claim chart or one or more other references identified in Defendants' Preliminary Invalidity Contentions ("Defendants' Invalidity Contentions").

To the extent a finder of fact determines that the references cited herein do not teach certain limitations in the asserted claims, such limitations would have been inherent and/or obvious. These claims are also invalid as obvious in view of *Kiss* alone or in combination with other prior art references, including, but not limited, to the prior art identified in the Cover Pleading of Defendants' Invalidity Contentions, the prior art described in the claim charts attached in Exhibit A, and/or the prior art identified in Exhibit D.

Defendants' Invalidity Contentions are based, in part, upon Defendants' present understanding of the asserted claims and IPA's apparent interpretations of the asserted claims in its July 10, 2019 Preliminary Infringement Contentions ("Infringement Contentions") and Defendants' investigation to date. Defendants are not adopting IPA's constructions or apparent constructions, nor are Defendants admitting to the accuracy of any particular contention or construction. The citations provided in the charts below are exemplary rather than exhaustive and Defendants reserve the right to rely upon additional references uncovered through further searching, other portions of the cited references and/or other portions of references cited within these Invalidity Contentions. Defendants further incorporate by reference the reservation of rights identified in the cover pleading to these Invalidity Contentions as though fully set forth herein.

[1] *Kiss* claims priority to provisional application 60/093,522 which was filed on July 21, 1998, before the January 5, 1999 earliest possible filling date of the '115 patent.

| | Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | with the capability to negotiate with each other, conduct joint planning, and to collaborate in the execution of planned tasks.<br><br>*Kiss* at 3:30-36.<br><br>*See also,* Fig. 1, Fig. 4, Fig. 6, 3:4-11, 6:45-7:19, 8:41-48, 10:32-35, 11:1-3, 11:37-38, 11:53-12:20. |
| 1(b) | a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events | A layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.<br><br>*Kiss* discloses other limitations of this claim as identified above. This limitation is obvious in view of *Kiss* combined with the knowledge of a person having ordinary skill in the art and/or any one or more of the references identified in the corresponding limitation of Ex. A-X. As Ex. A-X shows, each of these references also discloses this limitation. A person having ordinary skill in the art would have been motivated to combine *Kiss* with one or more of the references identified in the corresponding limitation of Ex. A-X rendering this limitation obvious based on one or more of the motivations to combine identified in § II.A.3.e of the cover pleading to Defendants' Preliminary Invalidity Contentions and because each of these references relate to the same field of software agent technology and distributed computing environments. |
| 1(c) | wherein the parameter lists further refine the one or more events; | Wherein the parameter lists further refine the one or more events.<br><br>*Kiss* discloses other limitations of this claim as identified above. This limitation is obvious in view of *Kiss* combined with the knowledge of a person having ordinary skill in the art and/or any one or more of the references identified in the corresponding limitation of Ex. A-X. As Ex. A-X shows, each of these references also discloses this limitation. A person having ordinary skill in the art would have been motivated to combine *Kiss* with one or more of the references identified in the corresponding limitation of Ex. A-X rendering this limitation obvious based on one or more of the motivations to combine identified in § II.A.3.e of the cover pleading to Defendants' Preliminary Invalidity Contentions and because each of these references relate to the |

**Exhibit A-X**

**Invalidity of U.S. Patent No. 6,851,115 ("'115 Patent")**
**by Secondary References**

As shown in the claim chart below, the asserted claims of the '115 patent are invalid under 35 U.S.C. § 102(a), 35 U.S.C. § 102(b), and/or 35 U.S.C. § 102(e) as anticipated and/or are invalid under 35 U.S.C. § 103 as obvious, or in combination with the reference(s) specifically identified in the following claim chart or one or more other references identified in Defendants' Preliminary Invalidity Contentions ("Defendants' Invalidity Contentions.").

The Secondary References relied upon herein include, but are not limited to:

- Bian, C., An Experimental Environment for Cooperative Agents, Thesis for the Master's Degree in Computer Science, Department of Computer Science Institute of Mathematical and Physical Sciences University of Tromso (June 30, 1997) ("Bian").
- Bian et al., "ViSe2 – An Agent-Based Expert Consulting System with Efficient Cooperation," Proceedings of the 1997 IEEE International Conference on Intelligent Engineering Systems (Sept. 15-17, 1997) ("Bian 2").
- Cheyer et al., "Multi-Modal Maps Using an Open Agent Architecture," Video, (1995) ("SFMAP").
- Farley, J., Java Distributed Computing, 1st ed. (1998) ("Farley").
- Malone et al., "Agents for Information Sharing and Coordination: A History and Some Reflections," chapter in Bradshaw, J. (ed.), Software Agents, 1997 ("Malone").
- Moran et al., "Multimodal User Interfaces in the Open Agent Architecture" published on January 6, 1997 ("*Moran reference*").
- Odubiyi, J. et al., SAIRE - A Scalable Agent-Based Information Retrieval Engine, Proceedings of the First International Conference on Autonomous Agents (Jan. 1997) ("Odubiyi").
- Schwartz et al., Cooperating Heterogenous Systems (1995) ("Schwartz95").
- U.S. Patent No. 6,088,689 to Kohn et al., filed November 29, 1995 and issued July 11, 2000 ("Kohn");
- U.S. Patent No. 5,706,406 to Pollock, filed May 22, 1995 and issued January 6, 1998 ("Pollock");
- U.S. Patent No. 6,029,174 to Sprenger et al., filed October 31, 1998 and issued February 22, 2000 ("Sprenger").
- U.S. Patent No. 5,855,009 to Garcia et al., filed July 31, 1992 and issued December 29, 1998 ("Garcia").
- U.S. Patent No. 7,028,312 to Merrick et al., filed March 23, 1999 and issued April 11, 2006, with an earliest priority date of March 23, 1998 ("Merrick").
- Open Agent Architecture software and associated documents ("OAA") (Ex. A-1 incorporated by reference herein), including but not limited to:
    - Adam Cheyer, "agentlib.c" (updated Nov. 10, 1996), *available at*

http://www.ai.sri.com/~oaa/distribution/distribv1/download/pc/oaa_c.zip (oaa_c\agentlib\agentlib.c) ("OAA Agentlib.c").

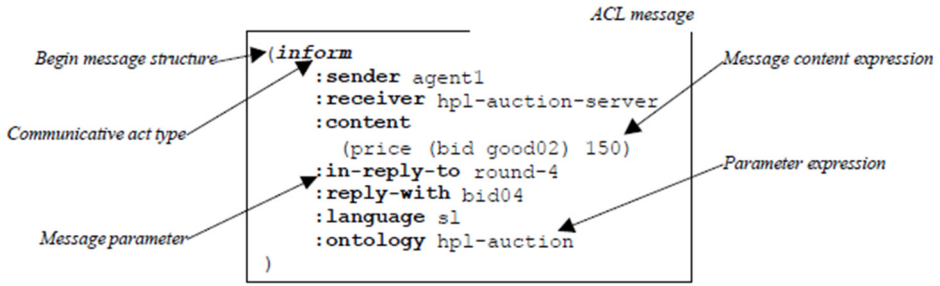
- "Documentation for Open Agent Architecture Agents" (July 19, 1995 to September 20, 1996), *available at* http://www.ai.sri.com/~oaa/distribution/agents/agents.html ("OAA Agents").
- "OAA Tutorial" (1994-1998), *available at* http://www.ai.sri.com/~oaa/distribution/distribv1/tutorial.html ("OAA Tutorial").

- The Open Agent ArchitectureTM – Building Communities of Distributed Software Agents presentation by Cheyer et al. published at least by February 21, 1998, available at http://www.ai.sri.com/~oaa/oaaslides/ ("Feb 1998 OAA Presentation") (Ex. A-2 incorporated by reference herein).
- OAA PAAM 1998 Presentation published at least by March 23, 1998 and disclosed prior to January 5, 1998 ("PAAM '98 Tutorial") (Ex. A-3 incorporated by reference herein).
- Genesereth, "An Agent-Based Framework of Interoperability" (1997) ("Genesereth '97") (Ex. A-7 incorporated by reference herein).
- Labrou, et al., "Semantics for an Agent Communication Language" (1996) ("Labrou") (Exs. A-7 and A-8 incorporated by reference herein).
- Singh, et al, "A Distributed and Autonomous Knowledge Sharing Approach to Software Interoperation" (March 22, 1995) ("Singh") (Ex. A-8 incorporated by reference herein).
- Lux, et al., "Understanding Cooperation: an Agent's Perspective" ("MECCA") (Ex. A-9 incorporated by reference herein).
- Cheyer, et al., "Multimodal Maps: An Agent-Based Approach" (June 9, 1995) ("Multimodal Maps Paper") (Ex. A-11 incorporated by reference herein).
- Finin, "Software Agents Knowledge Sharing KQML, KIF and Ontologies" (October 16, 1997), available at https://www.csee.umbc.edu/~finin/sisce97/ ("Finin I") (Ex. A-14 incorporated by reference herein).
- U.S. Patent No. 6,484,155 ("*Kiss*") filed on July 21, 1999, issued on November 19, 2002, with an earliest potential priority date of July 21, 1998 (Ex. A-15 incorporated by reference herein).
- "FIPA 97 Specification" by the Foundation for Intelligent Physical Agents ("*FIPA 97*") published on October 10, 1997 (Ex. A-16 incorporated by reference herein).
- Cheyer, et al, MVIEWS: Multimodal Tools for the Video Analyst" ("Cheyer") published at least by January 6, 1998 (Ex. A-17 incorporated by reference herein).
- Cohen, et al., "An Open Agent Architecture" ("Cohen") published on March 23, 1994 (Ex. A-18 incorporated by reference herein).
- Martin, et al., "Building Distributed Software Systems with the Open Agent Architecture," Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, March 23-25, 1998, London, UK (PAAM98) ("*Martin*") (Ex. A-19 incorporated by reference herein).

- InfoSleuth system and associated documents ("InfoSleuth") (Ex. A-20 incorporated by reference herein), including but not limited to:
    - Nodine et al., "Facilitating Open Communication in Agent Systems: The InfoSleuth Infrastructure," published in Intelligent Agents IV: Agent Theories, Architectures, and Languages, Proceedings 4th International Workshop, ATAL'97 (1998) ("Nodine").
    - Nodine et al., "Experience with the InfoSleuth Agent Architecture" published in the Proceedings of the Fifteenth National Conference on AI, September 27, 1998 ("Nodine 2").
    - Bayardo et al., "InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments," published in the Proceedings ACM SIGMOD International Conference on Management of Data, Vol. 26, Issue 2 (June 1997) ("Bayardo").
    - Urban et al., "Expressing Composite Events in InfoSleuth" published December 1998 ("Urban").
- General Magic system and associated documents ("General Magic") (Ex. A-21 incorporated by reference herein).
- RETSINA system and associated documents ("RETSINA") (Ex. A-22 incorporated by reference herein).
- Finin et al., "KQML as an Agent Communication Language," chapter in Bradshaw, J. (ed.), Software Agents, 1997 ("Finin II") (Ex. A-23 incorporated by reference herein).

To the extent a finder of fact determines that the references cited herein do not teach certain limitations in the asserted claims, such limitations would have been inherent and/or obvious. These claims are also invalid as obvious alone or in combination with other prior art references, including, but not limited, to the prior art identified in the Cover Pleading of Defendants' Invalidity Contentions, the prior art described in the claim charts attached in Appendix A, and/or the prior art identified in Appendix D.

Defendants' Invalidity Contentions are based, in part, upon Defendants' present understanding of the asserted claims and IPA's apparent interpretations of the asserted claims in its July 10, 2019 Preliminary Infringement Contentions ("Infringement Contentions") and Defendants' investigation to date. Defendants are not adopting IPA's constructions or apparent constructions, nor are Defendants admitting to the accuracy of any particular contention or construction. The citations provided in the charts below are exemplary rather than exhaustive and Defendants reserve the right to rely upon additional references uncovered through further searching, other portions of the cited references and/or other portions of references cited within these Invalidity Contentions. Defendants further incorporate by reference the reservation of rights identified in the cover pleading to these Invalidity Contentions as though fully set forth herein.

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | group, first of all, the agent exports its address and status information to the trader for registration, and imports all other cooperative agents' addresses into its cooperator-base; then the agent uses the imported addresses to set up cooperative connections with these agents; finally, if the users prefer their agents to work at cooperative states, the agents exchange their meta-level information of problem solving capabilities. After that, the agents are ready to serve their users as the all-round experts. |
| 1(b) | a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events | The Secondary References disclose a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events. *See, e.g.,:*<br><br>FIPA 97 discloses a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events. *See, e.g.,:*<br><br>The following figure summarises the main structural elements of an ACL message:<br><br>  <br>**Figure 1 — Components of a message**<br><br>In their transport form, messages are represented as s-expressions. The first element of the message is a word which identifies the communicative act being communicated, which defines the principal meaning of the message. There then follows a sequence of message parameters, introduced by parameter keywords beginning with a colon character. No space appears between the colon and the parameter keyword. One of the parameters contains the content of the message, encoded as an expression in some formalism (see below). Other parameters help the message transport service to deliver the message correctly (e.g. sender and |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | receiver), help the receiver to interpret the meaning of the message (e.g. language and ontology), or help the receiver to respond co-operatively (e.g. reply-with, reply-by).<br><br>*FIPA 97* at (FIPA97 Part 2): 12.<br><br>A message corresponds to a communicative act, in the sense that a message encodes the communicative act for reliable transmission between agents.<br><br>*FIPA 97* at (FIPA97 Part 2): 4.<br><br>For example, if i informs j that “Bonn is in Germany”, the content of the message from i to j is “Bonn is in Germany”, and the act is the act of informing. “Bonn is in Germany” has a certain meaning, and is true under any reasonable interpretation of the symbols “Bonn” and “Germany”, but the meaning of the message includes effects on (the mental attitudes of) agents i and j. The determination of this effect is essentially a private matter to both i and j, but for meaningful communication to take place, some reasonable expectations of those effects must be fulfilled.<br><br>*FIPA 97* at (FIPA97 Part 2): 11. |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|

Table 1 — Pre-defined message parameters

| Message Parameter: | Meaning: |
|---|---|
| `:sender` | Denotes the identity of the sender of the message, i.e. the name of the agent of the communicative act. |
| `:receiver` | Denotes the identity of the intended recipient of the message.<br><br>Note that the recipient may be a single agent name, or a tuple of agent names. This corresponds to the action of multicasting the message. Pragmatically, the semantics of this multicast is that the message is sent to each agent named in the tuple, and that the sender intends each of them to be recipient of the CA encoded in the message. For example, if an agent performs an inform act with a tuple of three agents as receiver, it denotes that the sender intends each of these agent to come to believe the content of the message. |
| `:content` | Denotes the content of the message; equivalently denotes the object of the action. |
| `:reply-with` | Introduces an *expression* which will be used by the agent responding to this message to identify the original message. Can be used to follow a conversation thread in a situation where multiple dialogues occur simultaneously.<br><br>E.g. if agent i sends to agent j a message which contains<br>`:reply-with query1,`<br>agent j will respond with a message containing<br>`:in-reply-to query1.` |
| `:in-reply-to` | Denotes an expression that references an earlier action to which this message is a reply. |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|

| | |
|---|---|
| :envelope | Denotes an expression that provides useful information about the message as seen by the message transport service. The content of this parameter is not defined in the specification, but may include time sent, time received, route, etc.<br><br>The structure of the envelope is a list of keyword value pairs, each of which denotes some aspect of the message service. |
| :language | Denotes the encoding scheme of the content of the action. |
| :ontology | Denotes the ontology which is used to give a meaning to the symbols in the content expression. |
| :reply-by | Denotes a time and/or date expression which indicates a guideline on the latest time by which the sending agent would like a reply. |
| :protocol | Introduces an identifier which denotes the *protocol* which the sending agent is employing. The protocol serves to give additional context for the interpretation of the message. Protocols are discussed in §7. |
| :conversation-id | Introduces an expression which is used to identify an ongoing sequence of communicative acts which together form a conversation. A conversation may be used by an agent to manage its communication strategies and activities. In addition the conversation may provide additional context for the interpretation of the meaning of a message. |

*FIPA 97* at (FIPA97 Part 2): 13-14.

*See also* 3, (FIPA97 Part 2): 11-22.

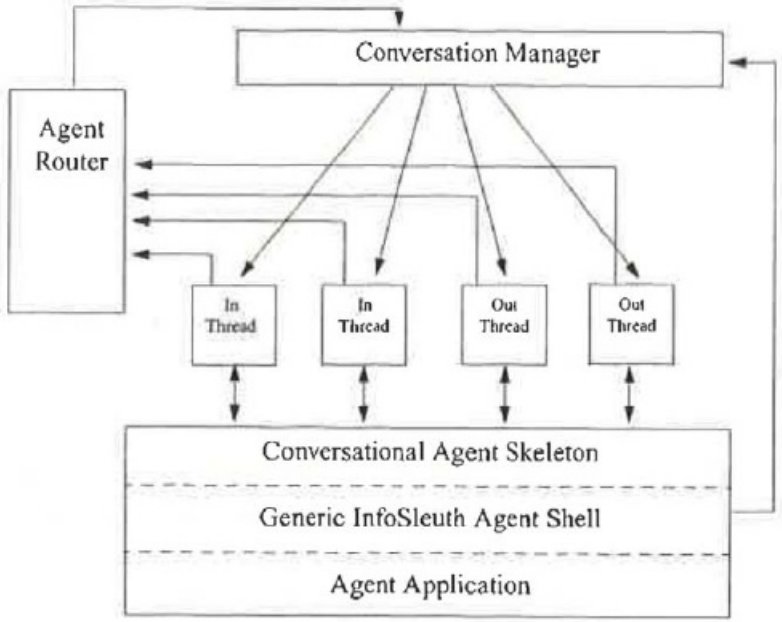
*Cohen* discloses “a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.”

*See, e.g., Cohen* at 3.

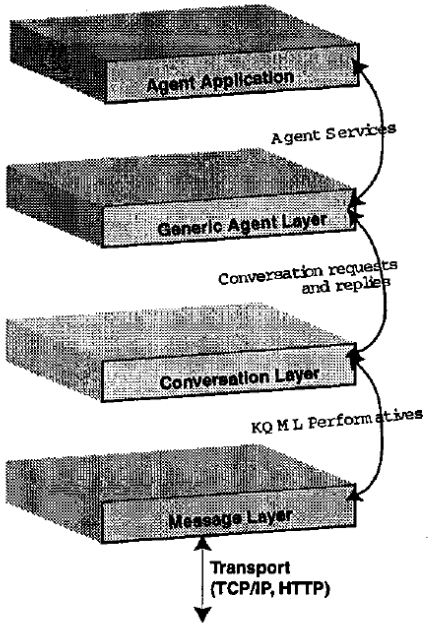
> The key to a functioning agent architecture is the interagent communication language. We explain ours in terms of its form and content. Regarding the

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | former, three speech act types are currently supported: Solve (i.e., a question), Do (a request) and Post (an assertion to the blackboard). For the time being, we have adopted little of the sophisticated semantics known to underlie such speech acts [5, 18, 19]. However, in attempting to protect an agent's internal state from being overwritten by uninvited information, we do not allow one agent to change another's internal state directly - only an agent that chooses to accept a speech act can do so. For example, a fact posted to the blackboard does not necessarily get placed in the database agent's files unless it so chooses, by placing a trigger on the black- board asking to be notified of certain changes in certain predicates (analogous to Apple Computer's Publish and Subscribe protocol).<br><br>*See also Cohen* at 2.<br><br>An agent consists of a Prolog meta-layer above a knowledge layer written in Prolog, C or Lisp. The knowledge layer, in turn, may lie on top of existing standalone applications (e.g. 'mailers, calendar programs, databases). The knowledge layer can access the functionality of the underlying application through the manipulation of files (e.g., mail spool, calendar datafiles), through calls to an application's API interface (e.g. MAPI in Microsoft Windows), through a scripting language, or through interpretation of an operating sys-tem's message events (Apple Events or Microsoft Windows Messages).<br><br>*See also Cohen* at 3-4.<br><br>Because delegated tasks and rules will be executed at distant times and places, users may not be able sim- ply to use direct manipulation techniques to select the items of interest, as those items may not yet exist, or their identities may be unknown. Rather, users will need to be able to describe arguments and invocation conditions, preferably in a natural language. Because these expressions will characterize events and their relationships, we expect natural language tense and aspect to be heavily employed [6]. Consequently, the meaning representation (or "logical form") produced by the multimodal interface will need to incorporate temporal information, which we do by |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | extending a Horn clause representation with time-indexed predicates and temporal constraints. The blackboard server will need to decompose these expressions, distribute pieces to the various relevant agents, and engage in temporal reasoning to determine if the appropriate constraints are satisfied.<br><br>With regard to the content of the language, we need to specify the language of predicates that will be shared among the agents. For example, if one agent needs to know the location of the user, it will post an expression, such as solve(location(user,U)), that another agent knows how to evaluate. Here, agreement among agents would be needed that the predicate name is location, and its arguments are a person and a location. The language of nonlogical predicates need not be fixed in advance, it need only be common. Achieving such commonality across developers and applications is among the goals of the ARPA "Knowledge Sharing Initiative," [13] and a similar effort is underway by the "Object Management Group" (OMG) CORBA initiative to determine a common set of objects.<br><br>*See also Cohen* at 1, 5-6.<br><br>InfoSleuth discloses a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events. *See, e.g.,*:<br><br>To implement prescriptive conversation policies in InfoSleuth, we have incorporated a conversation layer into our architecture.<br><br>Nodine at 291; Nodine at 290 (“the use of conversation protocols facilitates consistency in an open system by providing the agent with a definition of which message types it can expect and when”). |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>**Fig. 2.** Implementation of the InfoSleuth conversation layer.<br><br>Nodine at 292 (FIG. 2, illustrating “the InfoSleuth conversation layer”)<br><br>The conversation layer is implemented by several components: 1. a router, which dispatches messages to and receives messages from the KQML implementation layer; 2. a conversational agent skeleton, which dispatches messages to and receives messages from the local agent, and 3. a conversation manager, which coordinates both incoming and outgoing conversations via “in” and “out” conversation threads.<br><br>Nodine at 292; Nodine at 289-94 (section entitled “Conversation Policies”).<br><br>To implement prescriptive conversation policies in InfoSleuth, we have incorporated a conversation layer into our architecture. The conversation layer |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | defines and enforces the different conversations available to the agents in the InfoSleuth system, and is accessed via a clearly defined API. The conversation layer sits “‘on top of” the generic agent shell from which all our agents are being built. Thus, all InfoSleuth agents must communicate with other agents by using the conversation layer, which manages both incoming and outgoing conversations, and determines their legality.<br><br>Nodine at 291; Nodine at 290 (“the use of conversation protocols facilitates consistency in an open system by providing the agent with a definition of which message types it can expect and when”); Nodine at 290 (“conversation types should be uniquely identifiable by the types and/or parameters of the messages that initiate the conversation”).<br><br>The performatives each have a number of fields, or parameters associated with them — in addition to the content of the performative, other contextual parameters specify e.g., the language and ontology of the message.<br><br>Nodine at 285; Nodine at 285 (“build[ing] on KQML to define the set of speech acts used in our system”); *Id.* (“KQML . . . is an effort to define [a] standard useful set of speech acts, or performatives, that agents can use to exchange information . . . .”); Nodine at 291 (“In InfoSleuth, we currently define conversation policies based on the type of the initial performative.”) |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>Figure 2: InfoSleuth agent layers.<br><br>Nodine 2 at 66 (FIG. 2 showing “Conversational Layer”).<br><br>Conversation Layer The conversation layer of the InfoSleuth architecture defines and enforces conversation policies for a group of cooperating agents. A set of standard messages, e.g. in KQML, representing the available speech acts, can serve as a basis for very simple communication among agents. However, messages do not get sent in isolation; rather, there are often ongoing dialogs among two or more agents. Within a dialog, the interpretation of an individual message may depend on the context of the dialog in which they are participating. A “conversation” is a partially-ordered set of messages transmitted among a set of agents, all of which relate conceptually to an initiating speech act. A |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | conversation policy is a formal and deterministic specification of the ordering of speech acts exchanged between agents during a conversation. Conversation policies may be pres[c]riptive or emergent. A prescriptive conversation policy is one that is defined a priori, and used to enforce the agents’ communications. Emergent conversations are those in which the agents are not following specific external conversational policies; but rather where the performatives they use are determined by their internal functionality. These actions may be (but are not necessarily) driven by the agent’s semantic understanding of the discourse taking place (Smith & Cohen 1996).<br>Nodine 2 at 66.<br>We feel that KQML is effective when used compactly at the knowledge-level and agent systems should use existing protocols for streaming large information products between agents once they have agreed, by conversing in KQML, on the need to exchange an information product.<br>Nodine 2 at 69.<br>The DDAgent used the “subscription conversations” in the InfoSleuth agent shell to setup “encounter event notifications” with appropriate resources recommended by the broker. The pattern collection agent also used subscription conversations to setup “cost deviation notifications” from the DDAgent. For both of these analysis agents, the only work required was to interface the specific algorithms themselves to the event-based conversations in the agent shell. This application is quite intriguing because it demonstrates coordinated information gathering and analysis being performed at multiple levels (i.e., resources, multiresource norms, deviations, correlated patterns) and occurring in an event-driven manner in a dynamic network of information sources.<br>Nodine 2 at 70.<br>KQML (Knowledge Query and Manipulation Language) (Finin, Labrou, & Mayfield 1997) is an effort to define a standard useful set of speech acts, or |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | performatives, that agents can use to exchange information; as well as the (semi-formal) semantics behind the performatives.<br><br>Nodine 2 at 64.<br><br>The conversation layer of the InfoSleuth architecture defines and enforces conversation policies for a group of cooperating agents. A set of standard messages, e.g. in KQML, representing the available speech acts, can serve as a basis for very simple communication among agents. However, messages do not get sent in isolation; rather, there are often ongoing dialogs among two or more agents. Within a dialog, the interpretation of an individual message may depend on the context of the dialog in which they are participating. A “conversation” is a partially-ordered set of messages transmitted among a set of agents, all of which relate conceptually to an initiating speech act.<br><br>Nodine 2 at 66.<br><br>The performatives each have a number of fields, or parameters associated with them – in addition to the content of the performative, other contextual parameters specify e.g., the language and ontology of the message. Routing parameters specify the sender and receiver.<br><br>Nodine 2 at 64-65.<br><br>InfoSleuth provides a specialization of the agent shell termed the resource agent shell that encapsulates (and parameterizes) the conversations a resource may have with the other agents in the network.<br><br>Nodine 2 at 69.<br><br>This report describes the design of a composite event language for InfoSleuth. The language has adopted operators that have been introduced in the literature for composite event expression. We have enhanced the use of the operators with filters for expressing constraints on occurrences of different event combinations and for expressing temporal intervals on the event detection activities. One<br><br>Urban at Abstract; *see also* Urban at 10-11. |

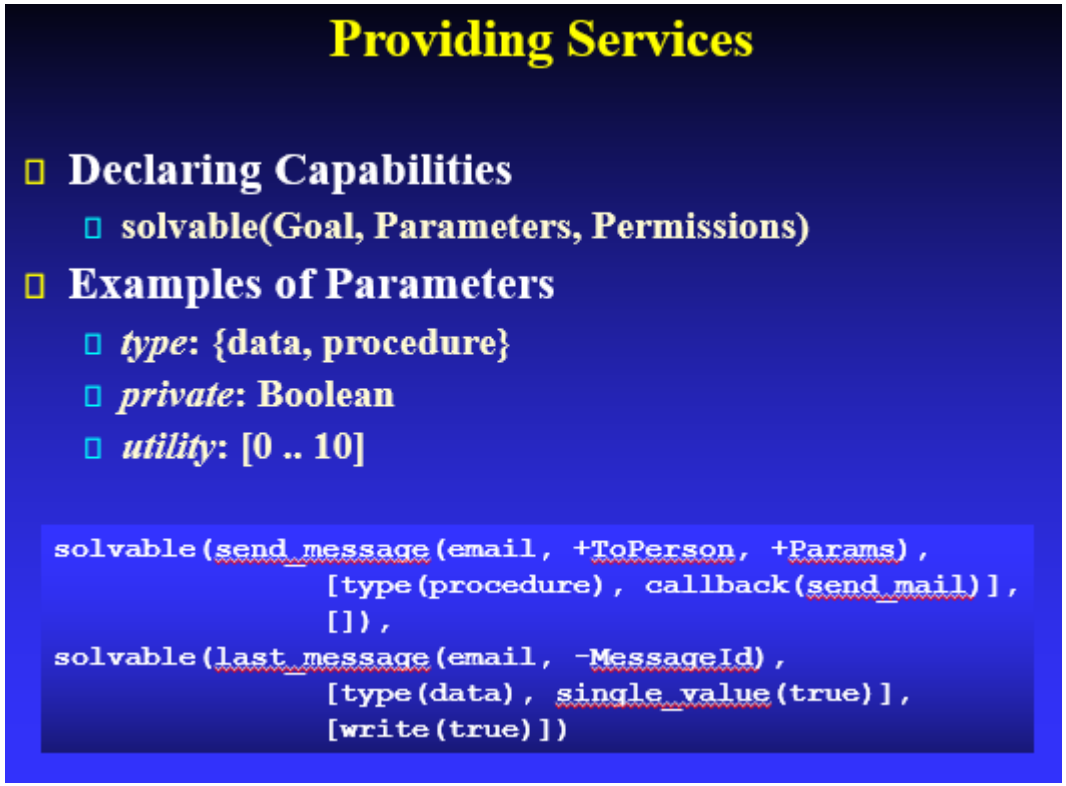
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | **3.2.2 Expressing the Composite Event**<br><br>Every data mining task generates events that correspond to each entry made in its output file. These events can be monitored by expressing a data mining query over the event ontology of InfoSleuth. Figure 5 illustrates the expression of a data mining query over the events of the deviation detection task in addition to data view queries that will be used to monitor news releases and publications. Events associated with these views (i.e., Assert on PatentDevs, Insert on News, and Insert on Pubs) are then used within a composite event expression that is submitted to InfoSleuth through the use of the MONITOR statement. This particular composite event is monitoring occurrences of patent deviations by a company, followed by at least several occurrences within three months of news articles or publications. These occurrences must occur within four months of the occurrence of the patent deviation event. The contents within square brackets represent constraints on the composite event, constraining news articles and publications to be related to the patent deviations through the company name, patent classification codes, and employees involved. The MONITOR statement also specifies the period during which the event detection is to take place, indicating that the event detection activity can include data associated with events that have occurred in the past.<br><br>Urban at 11, 15.<br><br>**4.1.3 View Update Event Filters**<br><br>Event filters can be used to define predicates on an event context. Predicates can be defined on either the user-defined parameters or the system-defined parameters. Filters are expressed by enhancing a view update event in the following manner: |

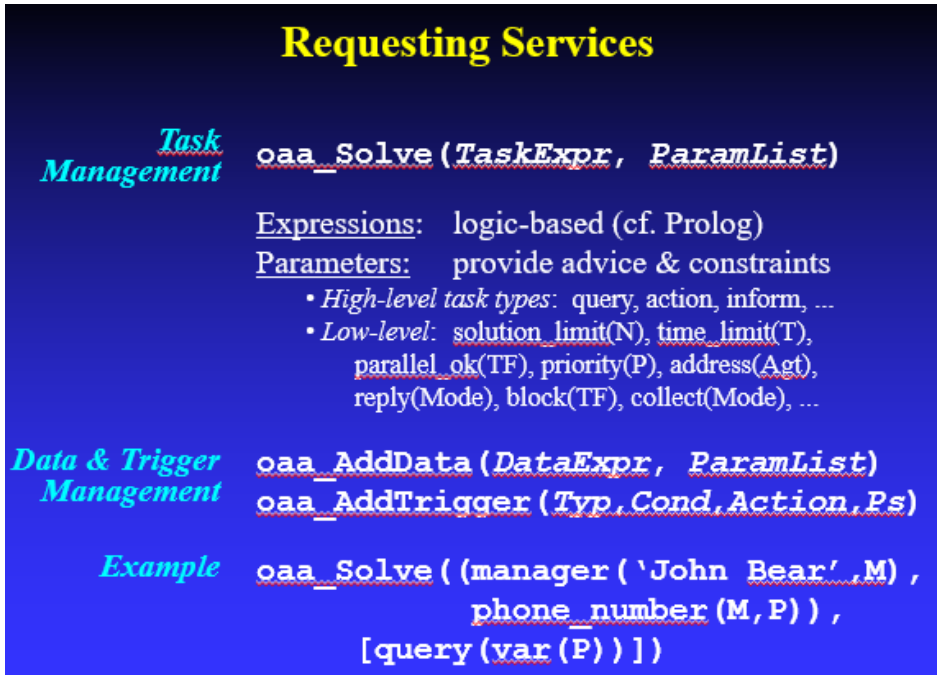
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | <*view-update-event*> on $S(p_1, p_2, ..., p_n)$ : [<*filter-predicate*>]<br><br>The <*filter-predicate*> is a constraint involving relational or set comparison terms expressed over the context of the event. AND, OR, and NOT operators can be used in the expression of the condition. The query subscription name is used to qualify all parameter names that appear within a filter expression (i.e., <*query_subscription_name*>.<*parameter_name*>). The syntax for the expression of terms must also be compatible with the general mode of condition expression within InfoSleuth. The following are examples of event filters over view update subscription names:<br><br>• Insert on DV-Query(X,Y,Z):[DV-Query.X > DV-Query.Y, DV-Query.Z = 10] - This filter defines a relational condition on the parameters of the event.<br><br>• Delete on DV-Query(X,Y,Z):[DV-Query.change-status = ”UPDATE”] - This filter refines the delete event to monitor only deletes that are caused by data updates in the resources currently serving the query.<br><br>• Insert on DV-Query(X,Y,Z):[DV-Query.change-status = ”ON-LINE” and DV-Query.tglobal > (08:00:00)10/01/98 and DV-Query.tglobal < (12:00:00)10/01/98] - This filter enhances the insert event to monitor only inserts that are caused by resources coming on line between the hours of 8 A.M. and Noon on October 1, 1998<br><br>• Assert on DM-Query(V,W):[”NY-Times-News” ∈ DM-Query.resource-set] - This filter defines a data mining event where the user is interested in monitoring events generated by data mining activities data resources involving news from the New York Times.<br><br>Urban at 15-16; *see also* Urban at 18-21.<br><br>Finin II discloses “a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.”<br><br>*See, e.g.,* Finin II at 347-48 (emphasis added).<br><br>The KQML language can be viewed as being divided into three layers: the content layer, the message layer and the **communication layer** . . . The communication layer encodes a set of features to the **message** which describe the lower level communication **parameters**, such as the identity of the sender and recipient, and a unique identi_er associated with the communication. |

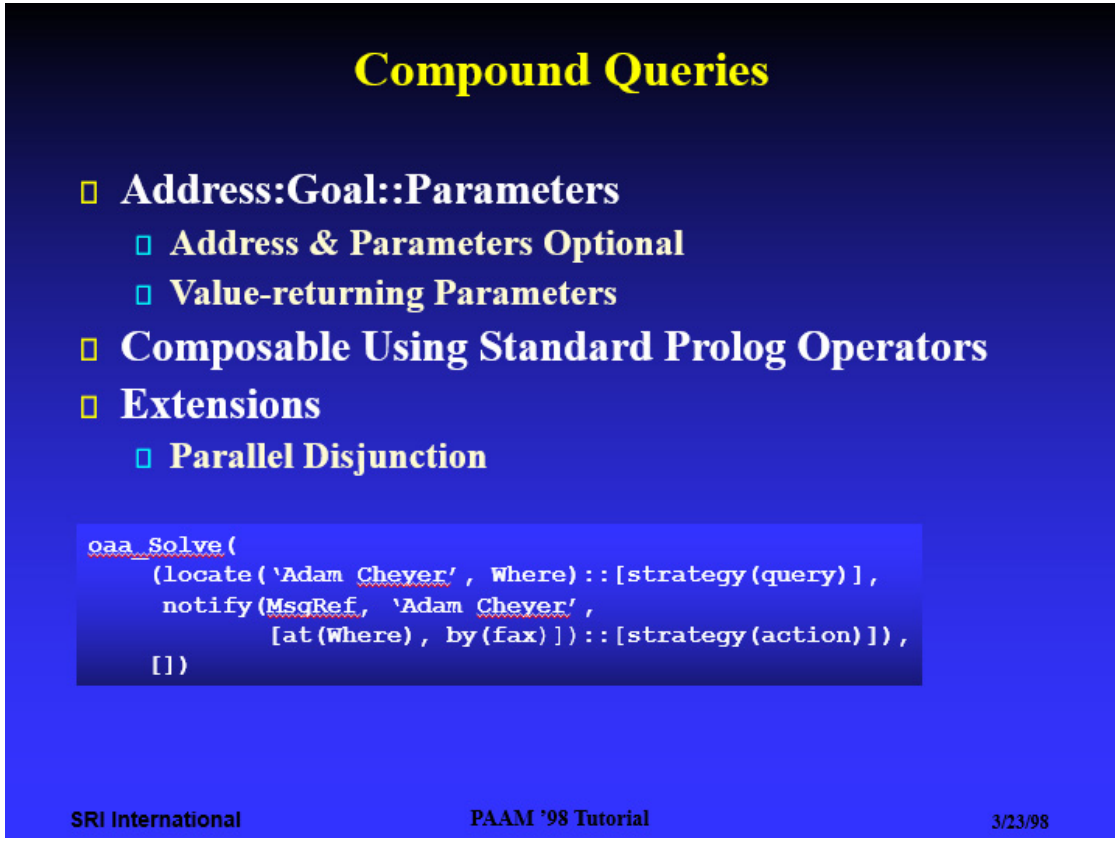
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | Merrick reference discloses a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.<br><br>*See, e.g.,* Merrick reference at 8:18-29.<br><br>> The invention is particularly applicable to remote procedure call (RPC), although it is applicable as well to other types of service invocation. The service invocation request and service invocation reply need not use the same message encoding, or even the same transfer protocol. The method of the invention, and particularly its message encoding, may be used with any of a number of transfer protocols, e.g., HTTP, FTP and SMTP, with its widest use at present being in conjunction with HTTP. In this context, the service to be invoked may be designated in the HTTP header, or in the URL. Alternatively, the service to be invoked may be identified within the message encoding.<br><br>*See also* Merrick reference at 12:14-28.<br><br>> The present assignee has developed XML RPC and a Business-to-Business Integration Server (B2B), while also enhancing WIDL to better support interaction with XML documents. XML RPC is an RPC mechanism that uses XML documents as the request and reply messages. A client machine sends a message to a server machine to ask the server machine to invoke a service. The server machine invokes the service and sends a reply message to the client. FIG. 1 illustrates the mechanism. In step 1, a client machine generates and transmits a request message to a server machine. In step 2, the server machine interprets the message as a request to invoke a service and invokes the service. In step 3, the server machine sends a reply message to the client machine. Step 3 is optional for the case where the client machine does not require a reply message.<br><br>Merrick reference at 13:66-14:19.<br><br>> An RPC protocol must include a transfer protocol. This application defines a transfer protocol as a mechanism for transferring messages between communicating programs. XML RPC may use any of a number of known transfer protocols, including network-specific protocols such as HTTP, SMTP, |

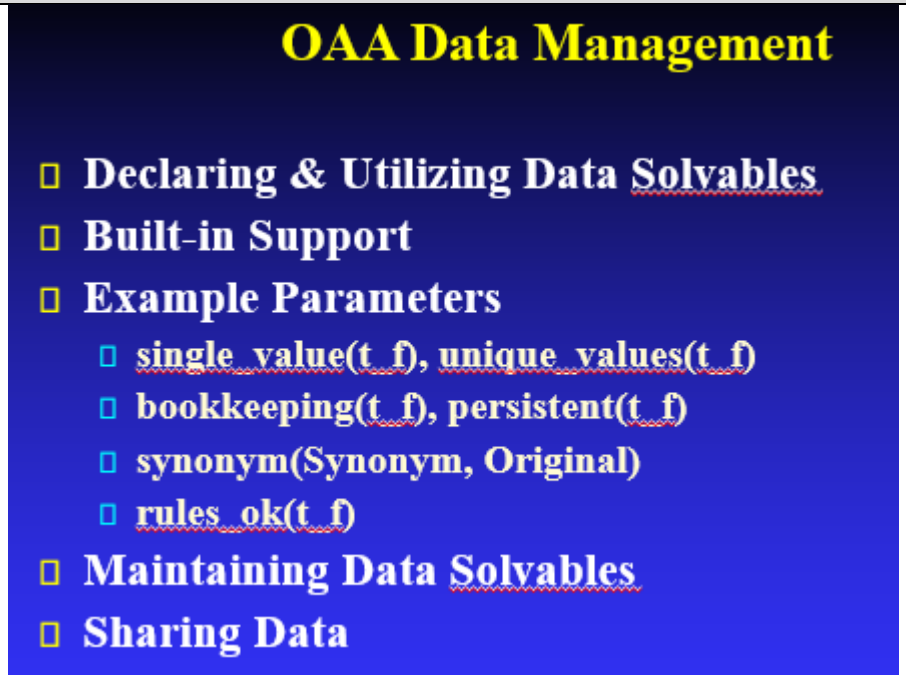
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | or FTP. FIG. 4 illustrates this process for the case where the transfer protocol is one of HTTP, SMTP, or FTP and the messages are transmitted across the Internet. According to the XML RPC procedure, and as depicted in step 1 of the figure, a client uses the transfer protocol to send an XML-based message to a server. This message is known as the request message. When HTTP is the transfer protocol, it is natural to accomplish this via HTTP's POST method. As shown in step 2, the server interprets the request message as a request to perform a particular service and performs the service. The server may perform this service using information found in the request message. When the service completes, the server generates a reply message. The reply message may contain information describing the results of the service. In step 3 the server then transmits the reply message to the client that initiated the request.<br><br>Merrick reference at 14:40-58.<br><br>One way to implement the RPC mechanism is to isolate the wire protocol layer from the service layer. One may isolate these layers on the client machine, on the server machine, or on both machines. FIG. 5 depicts an implementation in which the layers are isolated on the server machine. This application uses the term “integration server” to identify the layer that sits between the client machine and the service. In step 1, the client machine generates a request message and sends it to the integration server. The request message contains a set of input arguments. In step 2, the integration server extracts the arguments from the request message and passes the arguments to a service, invoking the service in the process. As shown in step 3, the service then performs an action. The action may be based on the input arguments that the integration server provided to it. In step 4, the service completes and returns output arguments to the integration server. The integration server generates a reply message that contains the output arguments and transmits the reply message to the client machine, as shown in step 5.<br><br>*See also* Merrick reference at 17:10-34; 18:40-19:6; 29:21-34:25. |

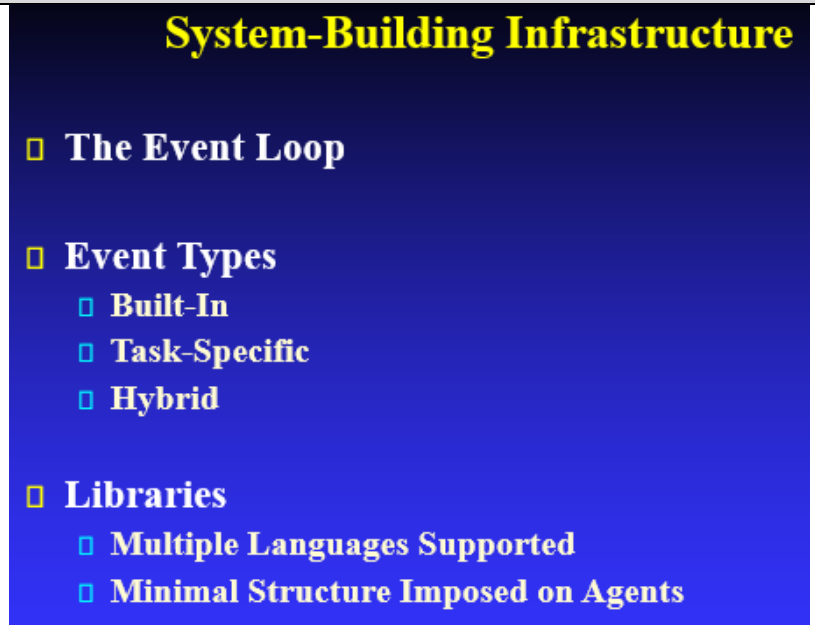
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | PAAM ’98 Tutorial discloses “a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.”<br><br>*See, e.g.,* PAAM ’98 Tutorial at p.27 (emphasis in original).<br><br>One of the most important program elements expressed in ICL is the *event*. The activities of every agent, as well as communications between agents, are structured around the transmission and handling of events. In communications, events serve as messages between agents; in regulating the activities of individual agents, they may be thought of as goals to be satisfied. Each event has a type, a set of parameters, and content. ICL includes a layer of conversational protocol (defined by the event types, together with the parameter lists).<br><br>  <br><br>*See also,* PAAM ’98 Tutorial at p.28 (emphasis in original). (Event is, e.g., “solvable” and parameters that refine the event are, e.g., “type: {data, procedure}, private: |

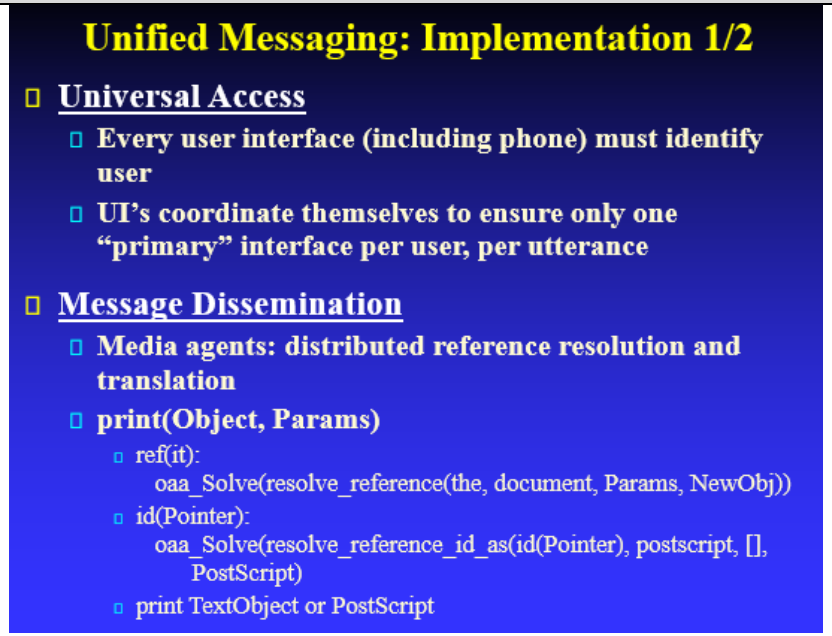
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | Boolean, utility: [0 . . .10]” listed under sub-bullets, and “[write(true)]” identified in the example).<br><br>Two major types of solvables are distinguished: *procedure* solvables and *data* solvables. Intuitively, a procedure solvable performs a test or action, whereas a data solvable provides access to a collection of data . . . A request for one of an agent's services normally arrives in the form of an *event* from the agent's facilitator. This event is then handled by the appropriate handler.<br><br>  <br><br>*See also,* PAAM ’98 Tutorial at p.29 (emphasis in original.) (Event is, for example, “oaa_Solve”, “oaa_AddData”, “oaa_AddTrigger” and parameters that refine the event |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | are, e.g., “High-level task types: query, action, inform, . . . ” “low-level: solution_limit(N), time_limit(T), parallel_ok(TF), priority(P), address(Agt), reply(Mode), block(TF), collect(Mode), …” listed under sub-bullets, and “[query(var(P))]” identified in the example.<br><br>An agent requests services by sending goals to its facilitator. Each goal contains calls to one or more solvables . . . The OAA libraries provide an agent with a single, unified point of entry for requesting services of other agents: the library procedure *oaa_Solve*.<br><br>  <br><br><br>*See also*, PAAM ’98 Tutorial at p.30 (Event is, for example, “oaa_Solve()” and parameters associated with and refining the event, including, for example, “[strategy(query)]” and “[strategy(action)]”). |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>*See also,* PAAM ’98 Tutorial at p.32 (emphasis in original).<br>[P]arameters [are] specified with the solvable's declaration. For example, the parameter single_value is used to indicate that the solvable should only contain a single fact at any given point in time. The parameter *unique_values* indicates that no duplicate values should be stored. Other parameters can allow data solvables to make use of the concepts of *ownership* and *persistence*. |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>*See also,* PAAM ’98 Tutorial at p.34. |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>*See also,* PAAM ’98 Tutorial at p.48 (Event is, for example, “oaa_Solve()” and parameters are, for example, “[],” identified in the example). |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>OAA discloses “a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.”<br>*See, e.g.,* OAA Tutorial at 3 (identifying messages in ICL with a conversation layer that identifies event types (e.g., “solve”) and parameters lists associated with one or more of the events, and refining the one or more events (e.g., “[solution_limit(1)]” among numerous other possible parameters)). |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|

**Tasking the Agent Community**

Agents may make requests of the agent community using the built-in primitive solve(). Solve() takes a goal to executed and a list of control parameters as arguments, and returns success, failure, or multiple solutions for th

In OAA 1.0, the argument list of control parameters may include:

- address(AgentAddress): which agent should solve the goal
- broadcast: just disseminate the message, do not request results returned.
- asynchronous: request solutions, but do not block. Solutions will be returned asynchronously in solved(FromAgent,Goal,Params,Solutions) messages which should be handled in do_event().
- solution_limit(N): limits the number of solutions found to N.
- time_limit(N): waits a maximum of N seconds before returning (failure if no solution found in given am time).
- level_limit(N): highest number of hierarchical Facilitator levels to climb looking for solutions.
- cache: cache all solutions locally and use cache before looking over the net.

Examples:

```
C:       solve("fax_number('Adam Cheyer', FaxNumber)", "[solution_limit(1)]", &answers)
Prolog:  solve(fax_number('Adam Cheyer', FaxNumber), [solution_limit(1)]).
```

*See also,* OAA Agents, p. 58 (identifying messages in ICL with a conversation layer, including, for example, “recognize_speech(Utterance)” where “solve” identifies the “event” and “[]” identifies the parameter list associated with and refining the event).

# recognize_speech(Utterance)

## Description

This command is used to request speech recognition in a blocking manner, i.e. with the command solve(recognize_speech(S), []).

*See also,* OAA Agents, p. 59 (identifying messages in ICL with a conversation layer, including, for example, “recognize_speech(Utterance,NLResult)” where “solve” identifies the “event” and “[]” identifies the parameter list associated with and refining the event).

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | **recognize_speech(Utterance,NLResult)**<br><br>**Description**<br><br>This command is used to request speech recognition in a blocking manner, i.e. with the command solve(recognize_speech(Str, NL), []).<br><br>*See also,* OAA Agents, p. 64 (identifying messages in ICL with a conversation layer, including, for example, “start(open_mic)” where “solve” identifies the “event” and “[broadcast]” identifies the parameter list associated with and refining the event).<br><br>**start(open_mic)**<br><br>**Description**<br><br>Start recognition in open mic mode, meaning that as soon as an utterance has been recognized, the speech agent will immediately start listening again.<br><br>This command is intended to be used in a non-blocking mode, i.e. solve(start(open_mic), [broadcast]).<br><br>*See also,* Agentlib.c, lines 1-79 (identifying messages in ICL with a conversation layer, including, for example, the event “solve(ID,Goal)” and “param_value” that holds the extracted values from the parameter list). |

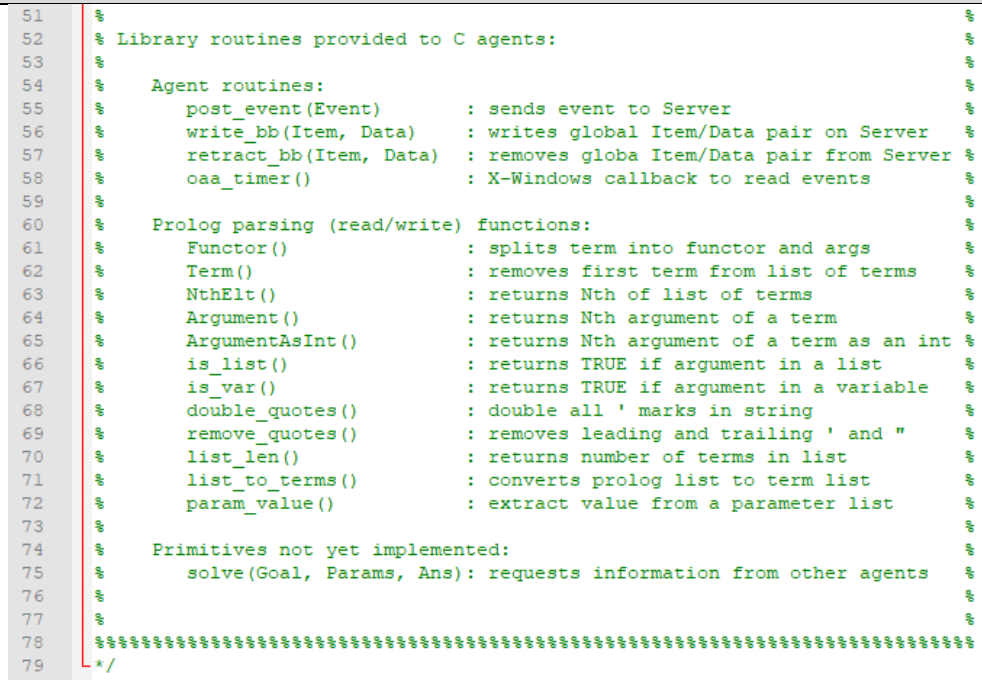
| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|

```
/* File    : agentlib.c
   Author  : Adam Cheyer
   Purpose : Contains C version of library for the Open Agent Architecture
   Updated : 11/10/96

   Copyright (C) 1995,1996 SRI International.  All rights reserved.

   "Open Agent Architecture" and "OAA" are trademarks of SRI International.
*/

/*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% AGENTLIB.c                                      Adam Cheyer     1/16/95      %
%                                                                              %
% Common routines used by all C agents in the Open Agent Architecture.         %
%                                                                              %
%                                                                              %
% Agent specific declarations:                                                 %
%                                                                              %
%       ksname                  : unique identifier, provided during           %
%                                 call to setup_communication()                %
%       timeout                 : timeout value for tcp_select, specified      %
%                                 by a call to set_timeout().                  %
%       solvable                : goals that the KS can solve. String          %
%                                 must be passed in as argument to             %
%                                 setup_communication.                         %
%                                                                              %
%       oaa_do_event()          : defines actions for agent-specific events%
%                                                                              %
%       oaa_app_idle()                  : to be executed if a timeout occurs
%                                                                              %
% Event requests that all agents can respond to:                               %
%                                                                              %
%       trace_on                : turns on event tracing                       %
%       trace_off               : turns off event tracing                      %
%       tcp_trace_on            : turns on tcp tracing                         %
%       tcp_trace_off           : turns off tcp tracing                        %
%       debug_on                : turns on event debugging                     %
%       debug_off               : turns off event debugging                    %
%       halt                    : quit to UNIX                                 %
%       solve(ID,Goal)          : tries to solve the goal, then returns an %
%                                 event solved(ID,FromKS, Goal, Solutions) %
%                                 to the caller.  ID is used to match the  %
%                                 calling goal to the solution.                %
%                                                                              %
% Events not yet implemented:                                                  %
%                                                                              %
%       set_timeout(Int, Mode)  : reinitializes the timeout for an agent   %
%       read_trigger(G,T,C,A)   : returns each operating trigger           %
```

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | |  <br>*See also,* Agentlib.c, lines 1229-1274 (identifying extracted values from a parameter list in the form “[p(val), p2(Val2)]” held in “param_value”). |

```
%                                                                             %
% Library routines provided to C agents:                                      %
%                                                                             %
%    Agent routines:                                                          %
%       post_event(Event)       : sends event to Server                       %
%       write_bb(Item, Data)    : writes global Item/Data pair on Server      %
%       retract_bb(Item, Data)  : removes globa Item/Data pair from Server    %
%       oaa_timer()             : X-Windows callback to read events           %
%                                                                             %
%    Prolog parsing (read/write) functions:                                   %
%       Functor()               : splits term into functor and args           %
%       Term()                  : removes first term from list of terms       %
%       NthElt()                : returns Nth of list of terms                %
%       Argument()              : returns Nth argument of a term              %
%       ArgumentAsInt()         : returns Nth argument of a term as an int    %
%       is_list()               : returns TRUE if argument in a list          %
%       is_var()                : returns TRUE if argument in a variable      %
%       double_quotes()         : double all ' marks in string                %
%       remove_quotes()         : removes leading and trailing ' and "        %
%       list_len()              : returns number of terms in list             %
%       list_to_terms()         : converts prolog list to term list           %
%       param_value()           : extract value from a parameter list         %
%                                                                             %
%    Primitives not yet implemented:                                          %
%       solve(Goal, Params, Ans): requests information from other agents      %
%                                                                             %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
*/
```

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|

```
1229 /*****************************************************************************
1230  * name:    param_value
1231  * purpose: Extracts the value from a parameter list
1232  *  inputs:
1233  *    - char *aList: a parameter list, in the form "[attr1(Value),attr2(Value2)]"
1234  *    - char *param: parameter to extract (e.g. attr1)
1235  * outputs:
1236  *    - char **Value: value of parameter or NULL if not found
1237  * remarks:
1238  *    - value will return a new copy of the argument, which should be
1239  *      be free'd when finished using.
1240  *    - most parameter lists should be in the form [p(Val), p2(Val2)]...
1241  *      If p has more than one argument, the list is returned
1242  *      If p has no arguments, "true" is returned as the value
1243  *****************************************************************************/
1244 EXPORT_MSCPP
1245 void EXPORT_BORLAND param_value(char *aList,
1246                                 char *param,
1247                                 char **value)
1248 {
1249    char *p;
1250    char *tmp = NULL;
1251    char *func = NULL;
1252    char *args = NULL;
1253    char *elt = NULL;
1254
1255    *value = NULL;
1256    tmp = strdup(aList);
1257    p = tmp;
1258    list_to_terms(tmp);
1259
1260    while (*tmp && !(*value)) {
1261       Term(tmp, &elt, &tmp);
1262       Functor(elt,&func,&args);
1263       if (strcmp(func, param) == 0) {
1264      if (args && *args)
1265         *value = strdup(args);
1266      else *value = strdup("true");
1267       }
1268       OAA_FREE(elt);
1269       OAA_FREE(args);
1270       OAA_FREE(func);
1271    }
1272
1273    OAA_FREE(p);
1274 }
```

*See also,* Agentlib.c, lines 2354-2435 (identifying messages in ICL with a conversation layer, including, for example, the “solve” event and numerous types of parameters associated with and refining the “solve” event (e.g., “level_limit(N),”

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | “address(KS),” “and_parallel,” “or_parallel,” “broadcast,” “solution_limit(N),” and “time_limit(N), among others)).<br>2354 /*****************************************************************************<br>2355 * name: solve<br>2356 * purpose: requests data or actions from distributed agents<br>2357 * inputs:<br>2358 * - char *goal: an ICL goal to be solve/performed<br>2359 * - char *params: control parameters describing how the goal should be<br>2360 * sent/resolved (see below).<br>2361 * outputs:<br>2362 * - Answers: a list containing solutions to the requested goal.<br>2363 * The answer value "[]" constitutes failure of the goal.<br>2364 * parameters:<br>2365 * The params argument is a list which may contain:<br>2366 *<br>2367 * cache : cache all solutions locally<br>2368 * level_limit(N) : highest number of levels to climb for<br>2369 * solutions.<br>2370 * address(KS) : ask a specific KS to solve goal<br>2371 * and_parallel : and-parallel solve of Goal list<br>2372 * or_parallel : or-parallel solve of Goal list<br>2373 * test(Test) : only solve goal on blackboards where Test<br>2374 * succeeds locally.<br>2375 * broadcast : just disseminate message, do not request<br>2376 * results.<br>2377 * asynchronous : solve will not block until solution returns,<br>2378 * the solutions will be returned asynchronously<br>2379 * in solved(FromKs,Goal,Params,Solutions) messages<br>2380 * which should be handled by in oaa_do_event()<br>2381 * solution_limit(N): limits the number of solutions found to<br>2382 * N. Currently only works for 1...<br>2383 * time_limit(N) : Waits a maximum of N seconds before<br>2384 * returning (failure if no solution).<br>2385 *<br>2386 * remarks:<br>2387 * If "broadcast" is sent in the parameter list, Ans may be sent as<br>2388 * NULL. Otherwise, solve will return anwers in a new space which<br>2389 * should be deallocated using OAA_FREE() when no longer needed.<br>2390 *****************************************************************************/<br>2391 EXPORT_MSCPP<br>2392 void EXPORT_BORLAND solve(char *Goal,<br>2393 char *Params,<br>2394 char **Ans)<br>2395 {<br>2396 char *event = NULL;<br>2397 char *solved_event = NULL;<br>2398<br>2399 /* If connected, perform solve request */<br>2400 #ifdef IS_WINDOWS<br>2401 if (Connection != INVALID_SOCKET) {<br>2402 #endif<br>2403 |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | 2404 #ifdef IS_UNIX<br>2405 if (Connection >= 0) {<br>2406 #endif<br>2407<br>2408 event = malloc(strlen(Goal) + strlen(Params) + 20);<br>2409 sprintf(event, "post_query(%s,%s)", Goal, Params);<br>2410 post_event(event);<br>2411<br>2412 OAA_FREE(event);<br>2413<br>2414 /* Broadcast and asynchronous not requested: wait for return event */<br>2415 if ((strstr(Params, "broadcast") == 0) &&<br>2416 (strstr(Params,"asynchronous") == 0)) {<br>2417 event = malloc(strlen(Goal) + 50);<br>2418 sprintf(event, "solved(_FromKS, %s, _Params, _Solutions)", Goal);<br>2419 poll_until_event(event, &solved_event);<br>2420 if (solved_event)<br>2421 Argument(solved_event, 4, Ans); /* Solutions */<br>2422 }<br>2423 /* Broadcast or Asynchronous requested:return immediately (no answers) */<br>2424 else {<br>2425 if (Ans)<br>2426 *Ans = strdup("[]");<br>2427 }<br>2428 OAA_FREE(event);<br>2429 OAA_FREE(solved_event);<br>2430 }<br>2431 /* if not connected, just return failure */<br>2432 else<br>2433 if (Ans)<br>2434 *Ans = strdup("[]");<br>2435 }<br>*Martin* discloses a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.<br>*See*, *e.g.*,:<br>The ICL includes a layer of conversational protocol, similar in spirit to that provided by KQML. and a content layer, analogous to that provided by KIF. The conversational layer of ICL is defined by the event types, together with the parameter lists that are associated with certain of these event types. . . . The conversational protocol makes use of an orthogonal, parameterized approach. That is, the conversational aspects of each element of an interagent |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | conversation are represented by a selection of an event type, in combination with a selection of values for an orthogonal set of parameters.<br><br>*Martin* at 363.<br><br>*See also*, *e.g.*,:<br><br>While it is possible to embed content expressed in other languages within an ICL event, it is advantageous to express content in ICL wherever possible. The primary reason for this is to allow the facilitator access to the content, as well as the conversational layer, in delegating requests.<br><br>*Id*.<br><br>Genesereth ’97 discloses “a layer of conversational protocol defined by event types and parameter lists associated with one or more of the events.”<br><br>*See, e.g.,* Genesereth ’97 at 322 (emphasis added).<br><br>ACL can best be thought of as consisting of three parts: an “inner” language called KIF (Knowledge Interchange Format), its vocabulary, and **an “outer” language called KQML (Knowledge Query and Manipulation Language)**. An ACL message is a KQML expression in which the “arguments’” are terms or sentences in KIF formed from words in the ACL vocabulary.<br><br>*See also,* Genesereth ’97 at 324.<br><br>**Knowledge Query and Manipulation Language (KQML)**<br><br>While it is possible to design an entire communication framework in which all messages take the form of KIF sentences, this would be inefficient. Because of the contextual independence of KIF's Labrou, each message would have to include any implicit information about the sender, the receiver, the time of the message, message history, and so forth. The efficiency of communication can be enhanced by providing a linguistic layer in which context is taken into account. This is the function of KQML. |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | *See also,* Genesereth ’97 at 224.<br><br>As used in ACL, each KQML *message* is a list of components enclosed in matching parentheses. The first word in the list indicates the type of communication. The subsequent entries are KIF expressions appropriate to that communication, in effect the “arguments.”<br><br>Intuitively, each message in KQML is one piece of a dialogue between the sender and the receiver, and KQML provides support for a wide variety of such dialogue types.<br><br>*See also,* Genesereth ’97 at 225.<br><br>The expression shown below is the simplest possible KQML dialog. In this case, there is just one message: a simple notification. The sender is conveying the enclosed sentence to the receiver. In general, there is no expectation on the sender's part about what use the receiver will make of this information.<br><br>A to B: (tell (> 3 2))<br><br>The following dialogue is a little more interesting. In this case, the first message is a request for the receiver to execute the operation of printing a string to its standard i/o stream. The second message tells the sender that the request has been satisfied.<br><br>A to B:(perform (print “Hello!” t))<br><br>B to A: (reply done)<br><br>In the dialogue shown below, the sender is asking the receiver a question in an ask-if message. The receiver then sends the answer to the original sender in a reply message.<br><br>A to B: (ask-if (> (size chip1) (size chip2)))<br><br>B to A: (reply true)<br><br>In the following case, the sender asks the receiver to send it a notification |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | whenever it receives information about the position of an object. The receiver sends it three such sentences, after which the original sender cancels the service.<br><br>A to B: (subscribe (position ?x ?r ?c))<br><br>B to A: (tell (position chip1 8 10))<br><br>B to A: (tell (position chip2 8 46))<br><br>B to A: (tell (position chip3 8 64))<br><br>A to B: (unsubscribe (position ?x ?r ?c))<br><br>In addition to simple notifications, commands, questions, and subscriptions, KQML also contains support for delayed and conditional operations, requests for bids, offers, promises, and so forth.<br><br>*See also,* Genesereth ’97 at 324 (identifying “ontology,” which is a parameter associated with and refining the event).<br><br>**Vocabulary**<br><br>In order for programs to communicate about an application area, they must use words that refer to the objects, functions, and relations appropriate to that application area, and they must use these words consistently. One way to promote this consistency is to create an open-ended dictionary of words appropriate to common application areas. Each word in the dictionary would have an English description for use by humans in understanding the meaning of the word, and each word would have KIF annotations for use by facilitators in mediating disagreements of terminology.<br><br>Note that, in proposing such a dictionary I am not proposing that there be one standard way of encoding information in each application area. Indeed, the dictionary would probably contain multiple *ontologies* for many areas. For example, we would expect it to contain vocabulary for describing three dimensional geometry in terms of polar coordinates, rectangular coordinates, cylindrical coordinates, and so on. Each program could use whichever ontology |

| | ’115 Patent Claim Language | Invalidity in View of Prior Art |
|---|---|---|
| | | is most convenient. The formal definitions of the words associated with any one of these ontologies could then be used by the facilitator in translating messages using one ontology into messages using the other ontology. These issues are discussed in more detail in the section on translation.<br><br>*See also,* Genesereth ’97 at 326 (identifying “ask-all” as the event type).<br><br>If some other agent $A_1$ wants to know the dealers of NEC, it may communicate the following request to the facilitator:<br><br>(ask-all ?x (dealer ?x nec))<br><br>The facilitator examines its knowledge base and determines that the business-agent can handle the request. The facilitator sends the request to the business-agent, gets the answer, and passes it to $A_1$. Agent $A_1$ is completely unaware of the sequence of steps performed in servicing its request.<br><br>*See also,* Genesereth ’97 at 327 (identifying “tell” as the event type).<br><br>This states that the cs-manager agent is interested only in the release of PC-compatible computers from IBM, Toshiba, NEC, and Micro-International. If another agent transmits the following fact to the facilitator:<br><br>(tell (released micro-international PC 6500D))<br><br>then the facilitator will examine its knowledge base and find that the agent cs-manager is interested in expressions of this form, and it will send the same KQML expression to the cs-manager.<br><br>*See also,* Genesereth ’97 at 333 (emphasis added).<br><br>Agents in a system may interoperate even when they are not created using the same programming language or development framework. Like programming “objects,” agents define message-based interfaces that are independent of their respective internal data structures and algorithms. The translation capability of facilitators extends this significantly by making interoperation independent of the agent interface (**the KQML expressions the agent can handle**). An agent |